Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC100010111001 — Programación Avanzada 11111100010-10

11001 de octubre de 11111100010

Actividad 1011

Manejo de *bytes*

Introducción

El malvado dictador Guilión, tirano de la localidad de Hammingtown, ha contactado al Dr. Herny y al mafioso Tini Tamburini, miembros de la liga DCC[1], pues ha sufrido un ataque informático perdiendo así el himno de su nación. Uno de los espías del dictador logró recuperar un par de archivos, junto con las instrucciones para recuperarlo. Dado que tienes altos conocimientos de manejo de *bytes*, se te pide que soluciones el dilema del dictador para mantener las relaciones diplomáticas de la liga DCC.

1. Búsqueda del archivo

Para comenzar con esta actividad, deberás ejecutar el archivo `generar_shrek.py`.

¡Sorpresa! Se crearon N carpetas `shrek` contenedoras de M carpetas `shrek` dentro de cada una. En alguna de estas $N \times M$ carpetas, se encuentra escondido el archivo `himno.shrek`, el cual deberás encontrar de forma **automática** (*i.e.* debes programar la solución). En el archivo `main.py` debes implementar la función `buscar_archivo(nombre)`, que recibe el nombre del archivo buscado y devuelve la ruta a dicho archivo.

Pista: La función `walk` del módulo *built-in* `os` te puede ayudar a realizar esta tarea de manera sencilla.

2. Decodificación

Luego de inspeccionar el archivo del himno, te das cuenta de que se encuentra corrupto —al igual que el dictador que lo compuso—, por lo que deberás repararlo utilizando un código de corrección de errores, que está descrito a continuación.

2.1. Lectura del archivo

Para poder aplicar el algoritmo de corrección de errores, debes primero leer el archivo y transformarlo a binario. Esto debes realizarlo completando la función `leer_archivo(ruta)` que recibe la ruta encontrada en la sección anterior. Para esto, deberás leer **cada dos *bytes*** del archivo del himno, y transformar cada uno de ellos en un arreglo en binario correspondiente. Cada arreglo binario obtenido debe contener **exactamente siete *bits***. Por la forma en que está construido el archivo se garantiza que los *bytes* que

[1] Claro, esto es el Departamento de Crueldad y Canalladas.

irás leyendo siempre pueden representarse con a lo más 7 bits. En caso de no tener dicho tamaño, debes rellenar con ceros a la izquierda[2]. Veamos un ejemplo.

$$24_{10} \rightarrow 11000_2 \rightarrow \texttt{11000} \rightarrow \texttt{0011000}$$
$$117_{10} \rightarrow 1110101_2 \rightarrow \texttt{1110101} \rightarrow \texttt{1110101}$$

Luego, debes unir ambas secuencias binarias formando una sola "palabra".

$$\texttt{0011000} + \texttt{1110101} \rightarrow \texttt{00110001110101}$$

Y finalmente, debes eliminar todos los *bits* de izquierda[3] a derecha hasta el primer `1` (incluyéndolo) que aparezca en esta palabra.

$$\texttt{00110001110101} \rightarrow \texttt{00110001110101} \rightarrow \texttt{10001110101}$$

Debe retornar una lista con las palabras que vaya formando y usarlas en el siguiente paso.

2.2. Algoritmo de decodificación

Tu misión es crear un decodificador Hamming[4] dentro de la función `decodificar(bits)`. Esta tomará una palabra de bits (obtenidas del paso anterior) y detectará si tiene un error en alguna posición, corregirá el bit (si hay error) y retornará los bits de datos. Este nos permite detectar errores de hasta un bit y nos entrega la posición en dónde ocurre el error.

Además, en este algoritmo se utilizan **bits de paridad**. Un bit de paridad es un bit que indica si en los siguientes bits existe una cantidad par o impar de `1` (unos). Si es par, se antepone un `0`; si es impar, un `1`. De tal forma que siempre que agreguemos un bit de paridad, el resultado tendrá una cantidad par de unos. Por ejemplo, `010110` tiene tres `1`, que es una cantidad impar; por lo tanto, su bit de paridad es `1`. Mientras que `101101` tiene una cantidad par (cuatro) de unos, por lo que su bit de paridad es `0`.

Bits originales	Con bit de paridad
`010110`	`1010110`
`101101`	`0101101`

En base a esto, debemos aplicar el algoritmo de corrección de errores a cada una de las palabras de bits que encontramos en el paso anterior. Por cada palabra deben:

1. Identificar los bits de paridad, que se encuentran en las posiciones $2^n - 1$, con $n \in \mathbb{N}$. Siguiendo con el ejemplo de la sección anterior, la palabra `10001110101`, al tener 11 bits, cuenta con **cuatro** bits de paridad (en las posiciones 0, 1, 3 y 7).

Posición	0	1	2	3	4	5	6	7	8	9	10
Palabra	1	0	0	0	1	1	1	0	1	0	1
	p1	p2		p3				p4			

2. Para cada bit de paridad, debes generar una nueva sub-palabra asociada a este bit:

   a) Tomar 2^i bits, incluyendo el bit de paridad.

[2] Aquí el uso de `zfill` es probablemente una buena idea.
[3] Y acá, el uso de `lstrip` también puede que sirva para eliminar, al menos, los ceros.
[4] Aquí tienes el enlace a un video explicativo de este algoritmo: `https://youtu.be/373FUw-2U2k`.

b) Saltar 2^i bits.

c) Repetir hasta que se terminen los bits.

Por ejemplo, para i con valor 0, se comienza en la posición $2^0 - 1$ y se debe tomar 2^0 bit (*i.e.* sólo el bit de paridad), luego saltarse 2^0 bit, tomar 2^0 bit, y seguir. Para el caso en que i tiene valor 1, se comienza de la posición $2^1 - 1$ y se toman 2^1 bits, luego se saltan 2^1 bits y así.

En esta tabla se puede ver un ejemplo de las palabras generadas por cada bit de paridad:

Posición	0	1	2	3	4	5	6	7	8	9	10	
	p1	p2		p3				p4				
Palabra	1	0	0	0	1	1	1	0	1	0	1	Sub-palabra obtenida
p1	1		0		1		1		1		1	`101111`
p2		0	0			1	1			0	1	`001101`
p3				0	1	1	1					`0111`
p4								0	1	0	1	`0101`

3. A cada sub-palabra obtenida se le debe calcular su propio bit de paridad, a estos bits se les llama bit de posición. En el ejemplo, los bits de posición para las sub-palabras obtenidas serían:

Paridad	Palabra obtenida	¿Par?	Posición
p1	`101111`	`False`	1
p2	`001101`	`False`	1
p3	`0111`	`False`	1
p4	`0101`	`True`	0

Los bits de posición formarán un número binario que codifica la posición del bit erróneo. Este se lee desde la paridad más alta a la más baja y se le debe restar uno. En el caso anterior, el número binario generado es `0111` (bit de posición de p4 & p3 & p2 & p1).

En caso de que todos los bits de posición sean 0's significa que en la palabra no hay ningún error.

Como el valor obtenido no posee sólo 0's, se debe transformar a su valor decimal y restar uno. El número 0111_2 en base 10 (decimal) es 7_{10}, por lo tanto el bit en la posición $7 - 1 = 6$ es el que debe ser corregido.

4. Finalmente, se debe corregir la palabra de bits obtenida (intercambiando el valor del bit erróneo) y eliminar todos los bits de paridad de la palabra.

El resultado del ejemplo es `0110101`, como se indica en la tabla siguiente.

Posición	0	1	2	3	4	5	6	7	8	9	10
	p1	p2		p3				p4			
Palabra errada	1	0	0	0	1	1	1	0	1	0	1
Palabra corregida	1	0	0	0	1	1	0	0	1	0	1
Datos obtenidos			0		1	1	0		1	0	1

2.3. Escritura del archivo decodificado

Finalmente debes llenar la función `escribir_archivo(ruta, chunks)`. Esta recibe la ruta y un arreglo de *bytes* y debe escribir los *bytes* en la ruta especificada. Debes tomar cada uno de los números binarios

obtenidos de la decodificación, transformarlos a su representación en *bytes* y escribirlos en el archivo `mapa.png`. En este archivo encontrarás la ubicación del himno del malvado dictador Guilión.

Entrega

- **Lugar:** En su repositorio privado de GitHub, en la **carpeta** `Actividades/AC1011/`
- **Hora del último *push*:** 10000:1010 (claro, 16:10)